A Brief Introduction to RenderMan

From Sig2006

Table of contents

1 Origins
2 Spec
3 Pipeline (RIBS, shaders, maps)
    3.1 RIB files
    3.2 Shaders
    3.3 Maps
4 RIB - syntax, semantics
5 Shader writing
6 Resources
    6.1 Books
    6.2 Notes
    6.3 Forums, portals

Origins

Pixar's RenderMan software has its origins in the University of Utah during the 1970s, where Pixar founder Ed Catmull did his PhD work on rendering problems. From there, the scene shifted to George Lucas' Lucasfilm in California, where Catmull and few other graphics researchers were brought in to work on graphics software specifically for use in motion pictures (a part of Lucasfilm later became Pixar).

The researchers had the explicit goal of being able to create complex, high quality photorealistic imagery, which were by definition virtually indistinguishable from filmed live action images. They began to create a renderer to help them achieve this audacious goal. The renderer had an innovative architecture designed from scratch, incorporating technical knowledge gained from past research both at Utah and NYIT. Loren Carpenter implemented core pieces of the rendering system, and Rob Cook wrote the shading subsystem. Pat Hanrahan served as the lead architect for the entire project. The rendering algorithm was termed “REYES”, a name with dual origins. It was inspired by Point Reyes, a picturesque spot on the California coastline which Carpenter loved to visit. To the rendering team the name was also an acronym for “Render Everything You Ever Saw”, a convenient phrase to sum up their ambitious undertaking.

At the 1987 SIGGRAPH conference, Cook, Carpenter and Catmull presented a paper called “The Reyes Rendering Architecture” which explained how the renderer functioned. Later at the SIGGRAPH in 1990, the shading language was presented in a paper titled “A Language for Shading and Lighting Calculations” by Hanrahan and Jim Lawson. In 1989 the software came to be known as RenderMan and began to be licensed to CG visual effects and animation companies. Also, the CG division of Lucasfilm was spun off into its own company, Pixar in 1983 and was purchased by Steve Jobs in 1986. The rest, as they say, is history..

Even though the public offering of RenderMan did not happen until 1989, the software was used internally at Lucasfilm/ Pixar way before that, to create movie visual effects, animation shorts and television commercials.

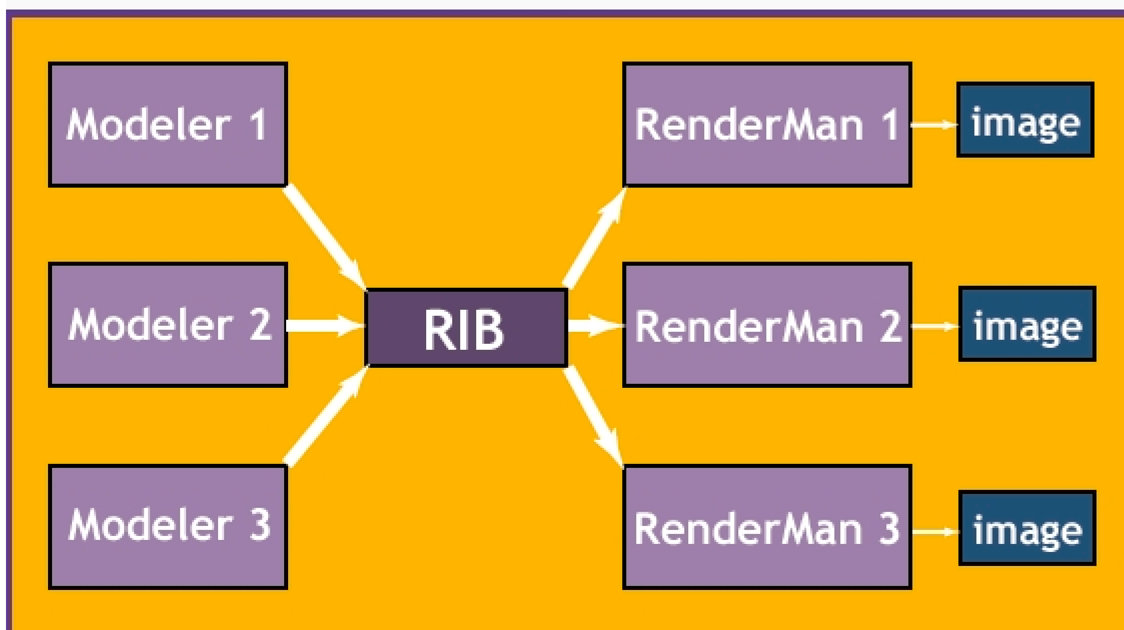
In 1982, the Genesis Effect in the movie Star Trek II: The Wrath of Khan was created using an early version of RenderMan, as was the stained glass knight in the movie Young Sherlock Holmes released in 1985.

Today, leading animation and visual effects studios around the world routinely use Pixar's RenderMan thanks to its unsurpassed track record - it is fast, stable, efficient when it comes to handling large scenes with complex geometry, surface appearances and lighting. The output is high quality photoreal imagery, usable on its own (eg. in animated features) or ready for compositing with existing footage (eg. in live-action movies).

Spec

In the 'Origins' section above, we began by referring to 'Pixar's RenderMan'. This is because, strictly speaking, the word 'RenderMan' by itself denotes an interface description originated by Pixar, to provide a standard way for modeling/animation programs to communicate their scene descriptions to renderers. In other words, RenderMan is a formal specification. It is referred to as the 'RI Spec', where 'RI' stands for 'RenderMan Interface'. Pixar's own implementation of the specification was historically the very first one, so people loosely refer to it (the implementation) also as 'RenderMan'. The correct name for Pixar's version is 'PRMan' (short for Photorealistic RenderMan), and this is the name we will use for it from now on.

In a 3D graphics pipeline, rendering is the last step (after modeling, animation and lighting) that generates an image out of a scene description. Renderers are specialized, complex programs that embody a variety of algorithms which collectively lead to image synthesis. The RI Spec defines a clear separation (or boundary, or interface) between modeling and animation programs on one hand, and rendering programs on the other. The idea is that each side can focus on its own specialty, and formal 'handshake' protocol can lead to successful mixing and matching between the two. In practical terms, this means that if a piece of modeling/animation program were to output its scene description in an established format, that scene description should be able to serve as input to a variety of renderers that handle that format. All the renderers would produce pretty much the same output image from the scene description, regardless of how their internals are structured. This is because the interface specifies what to render (via geometry, lights, material and camera descriptions) but not how. The 'how' is up to the individual implementations to handle - they can freely employ scanline algorithms, ray-tracing, radiosity, point-based graphics or any other technique to render the output.



The Spec was authored by Pixar and was endorsed by leading graphics vendors at the time such as Sun, Apollo, Prime and NeXT. The hope was that the Spec would give rise to a variety of implementations. As a proof of concept and to seed the marketplace, Pixar themselves created PRMan, the first-ever RenderMan implementation. Shortly thereafter, Larry

Gritz wrote his freeware BMRT which also adhered to the Spec. As a case in point, BMRT featured ray-tracing and radiosity which were only recently (in 2002) added to PRMan. RenderDotC from DotC Software was also an early implementation which continues to be sold to this day.

Fast-forwarding to more recent times, there have been several other RenderMan implementations since the early days. Exluna Corporation (founded by Larry Gritz and others) created and sold Entropy, a commercial version of BMRT (however, due to an unfortunate lawsuit, Exluna was shut down and Entropy/BMRT were taken off the market). Air, Aqsis, Angel, Pixie and 3Delight are contemporary implementations which should be of interest to attendees of this course. Together with RenderDotC, they provide alternatives to PRMan. While PRMan remains the industry's gold standard for RenderMan implementations, it also happens to be more expensive than the alternatives (many of which are free!).

Here is a brief tour of the Spec, which is divided into two parts. Part I, 'The RenderMan Interface', begins by listing the core capabilities (required features) that all RenderMan-compliant renderers need to provide, such as a complete hierarchical graphics state, camera transformations, pixel filtering and antialiasing and the ability to do shading calculations via user-supplied shaders written in the RenderMan shading language. This is followed by a list of advanced/optional capabilities such as motion blur, depth of field and global illumination. The interface is then described in great detail, using procedural API calls in C/C++ and their corresponding RIB (RenderMan Interface Bytestream) equivalents. RIB can be regarded as a scene description format meant for use by modeling programs to generate data for RenderMan-compliant renderers. Part II of the Spec, 'The RenderMan Shading Language' (RSL), describes a C-like language (with a rich set of shading-related function calls) for writing custom shading and lighting programs called *shaders*. This programmable shading aspect is one of the things that makes RenderMan enormously popular, since it gives users total control over lighting and appearances of surfaces and volumes in their scenes.

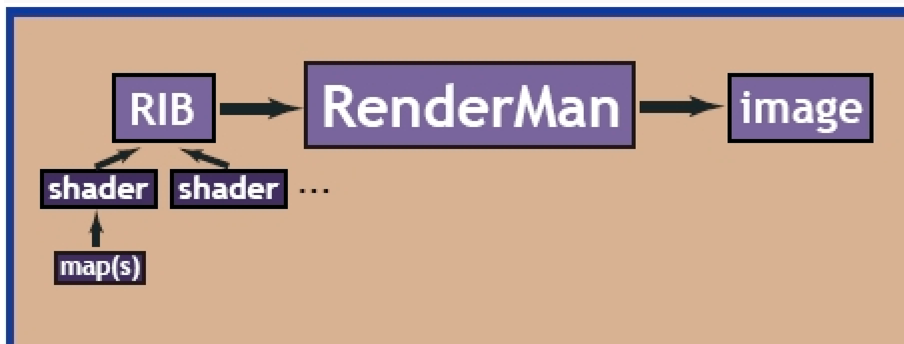
The latest version of the Spec is 3.2.1, revised in November 2005. You can find the 688K, 226 page document (which happens to make for enjoyable reading!) at Pixar's site: https://renderman.pixar.com/products/rispec/index.htm.

Be sure to get a good understanding of what is in the RI Spec - it will help you know what to expect in a typical RenderMan implementation (any renderer that calls itself 'RenderMan-compliant' will by definition be bound by the interface laid out in the Spec). In addition the Spec will serve as your reference for RIB and procedural API syntax and also for the large set of built-in functions of the RSL.

Pipeline (RIBS, shaders, maps)

In this section we will look at how you create RenderMan images in practice, using your renderer of choice (PRMan/RenderDotC/Air/Aqsis/Angel/Pixie/3Delight). While the overall pipeline is same for all these renderers, you will need to consult your particular renderer's manual for implementation-dependent details and minor syntax variations in RIB, etc.

Here is the overall flow of data to generate (render) output:

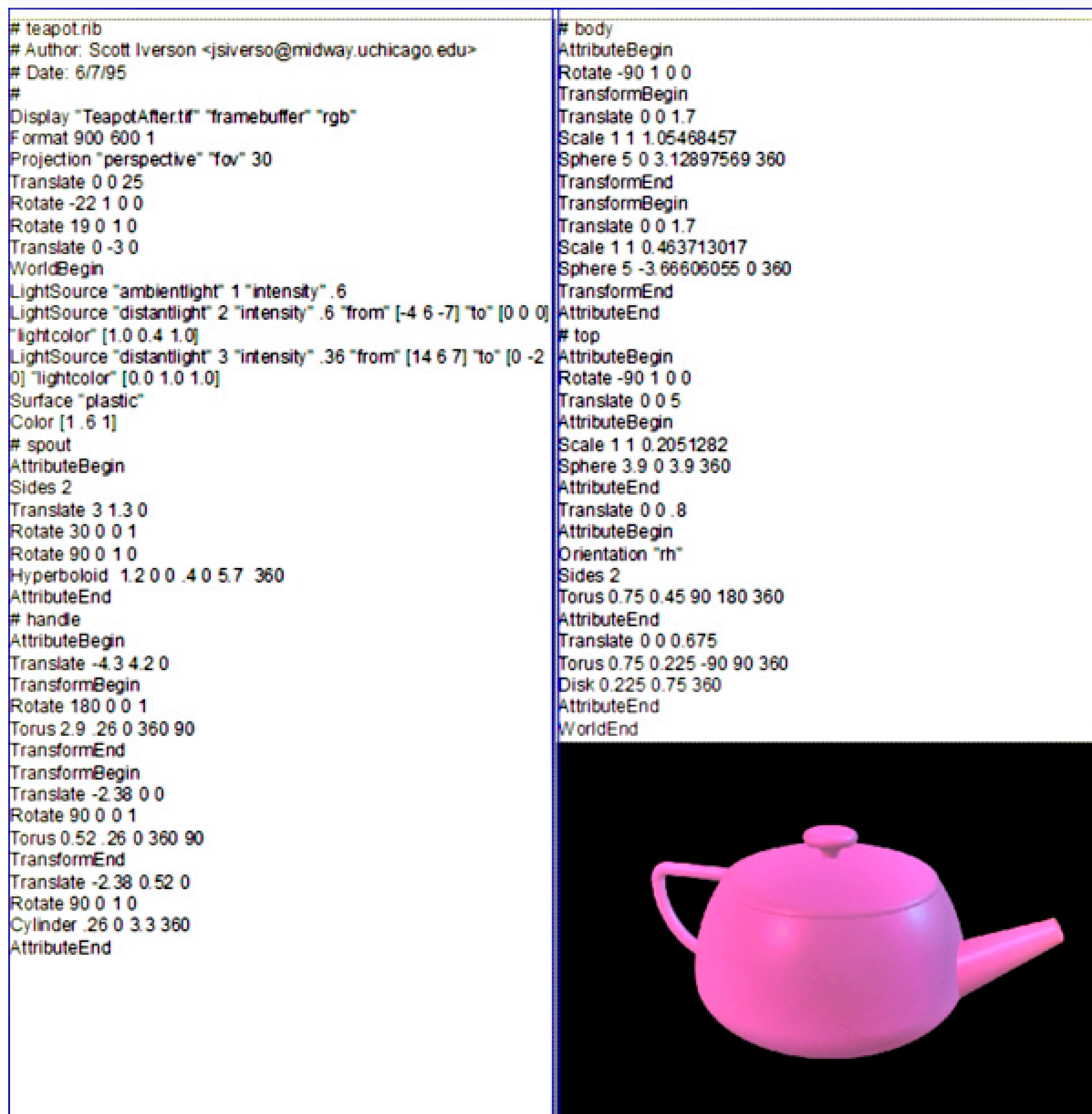


The renderer accepts a RIB file containing scene description that can be in ASCII or binary format, usually containing data for rendering just one frame. It reads the RIB file and renders a result image. The output image can either be displayed on a screen window or written to an image file, eg. in TIFF format. The RIB file will most likely contain references to shaders that describe shading, displacements or light sources. Such shader files are themselves external to the RIB file. Additionally, shaders can sometimes reference map files (eg. texture maps) which are in turn external to the shaders. The combination of a RIB file and its associated shaders and maps is what gets rendered into an output image.

The RIB file is input to the renderer via a simple command line invocation. Eg. in PRMan, the call would look like this:

render teapot.rib

'render' invokes PRMan, which will read 'teapot.rib' and render an output image according to the scene described in 'teapot.rib'.



```
# teapot.rib
# Author: Scott Iverson <jsiverso@midway.uchicago.edu>
# Date: 6/7/95
#
Display "TeapotAfter.tif" "framebuffer" "rgb"
Format 900 600 1
Projection "perspective" "fov" 30
Translate 0 0 25
Rotate -22 1 0 0
Rotate 19 0 1 0
Translate 0 -3 0
WorldBegin
LightSource "ambientlight" 1 "intensity" .6
LightSource "distantlight" 2 "intensity" .6 "from" [-4 6 -7] "to" [0 0 0]
"lightcolor" [1.0 0.4 1.0]
LightSource "distantlight" 3 "intensity" .36 "from" [14 6 7] "to" [0 -2
0] "lightcolor" [0.0 1.0 1.0]
Surface "plastic"
Color [1 .6 1]
# spout
AttributeBegin
Sides 2
Translate 3 1.3 0
Rotate 30 0 0 1
Rotate 90 0 1 0
Hyperboloid  1.2 0 0 .4 0 5.7  360
AttributeEnd
# handle
AttributeBegin
Translate -4.3 4.2 0
TransformBegin
Rotate 180 0 0 1
Torus 2.9 .26 0 360 90
TransformEnd
TransformBegin
Translate -2.38 0 0
Rotate 90 0 0 1
Torus 0.52 .26 0 360 90
TransformEnd
Translate -2.38 0.52 0
Rotate 90 0 1 0
Cylinder .26 0 3.3 360
AttributeEnd
# body
AttributeBegin
Rotate -90 1 0 0
TransformBegin
Translate 0 0 1.7
Scale 1 1 1.05468457
Sphere 5 0 3.12897569 360
TransformEnd
TransformBegin
Translate 0 0 1.7
Scale 1 1 0.463713017
Sphere 5 -3.66606055 0 360
TransformEnd
AttributeEnd
# top
AttributeBegin
Rotate -90 1 0 0
Translate 0 0 5
AttributeBegin
Scale 1 1 0.2051282
Sphere 3.9 0 3.9 360
AttributeEnd
Translate 0 0 .8
AttributeBegin
Orientation "rh"
Sides 2
Torus 0.75 0.45 90 180 360
AttributeEnd
Translate 0 0 0.675
Torus 0.75 0.225 -90 90 360
Disk 0.225 0.75 360
AttributeEnd
WorldEnd
```

Let us now briefly look at some characteristics of RIB files, shaders and maps.

RIB files

Sources of RIBs include the following:

- executable programs created using the procedural API calls from the Spec. When such a program is run, its output will be RIB statements, which can be redirected to a file for submitting to the renderer. Eg. a DNA rendering program would contain RI calls to create spheres of various sizes and colors, placed at specific locations as dictated by the DNA molecular structure.
- translator plugins that are part of mainstream animation packages which create RIB descriptions of scenes. These plugins would make RI calls corresponding to scene elements. Eg. for Maya, MTOR, MayaMan and Liquid are all plugins that output RIB.
- converter programs that read scene descriptions for other renderers and create corresponding RIB calls. Eg. 'mi2rib' is a Mental Ray to RenderMan converter.
- several modeling/animation programs (eg. Blender) natively output RIB, ie. without a translator plugin.
- simple RIB files can be hand-generated by the user, or output using scripting languages such as MEL. The RI API calls are bypassed, and RIB statements are directly output by the script, using appropriate syntax for each RIB call.

Here is a comparison (taken from 'RenderMan for Poets') that shows a C program that outputs RIB and the corresponding RIB statements.

```
#include <math.h>
#include "ri.h"
void main (void)
{
static RtFloat fov = 45, intensity = 0.5;
static RtFloat Ka = 0.5, Kd = 0.8, Ks = 0.2;
static RtPoint from = {0,0,1}, to = {0,10,0};
RiBegin (RI_NULL);
RiFormat (512, 512, 1);
RiPixelSamples (2, 2);
RiFrameBegin (1);
RiDisplay ("t1.tif", "file", "rgb", RI_NULL);
RiProjection ("perspective", "fov", &fov, RI_NULL);
RiTranslate (0, -1.5, 10);
RiRotate (-90, 1, 0, 0);
RiRotate (-10, 0, 1, 0);
RiWorldBegin ();
RiLightSource ("ambientlight", "intensity", &intensity,RI_NULL);
RiLightSource ("distantlight", "from", from, "to", to, RI_NULL);
RiSurface ("plastic", "Ka", &Ka, "Kd", &Kd, "Ks", &Ks, RI_NULL);
RiTranslate (.5, .5, .8);
RiSphere (5, -5, 5, 360, RI_NULL);
RiWorldEnd ();
RiFrameEnd ();
RiFrameBegin (2);
RiDisplay ("t2.tif", "file", "rgb", RI_NULL);
RiProjection ("perspective", "fov", &fov, RI_NULL);
RiTranslate (0, -2, 10);
RiRotate (-90, 1, 0, 0);
RiRotate (-20, 0, 1, 0);
RiWorldBegin ();
RiLightSource ("ambientlight", "intensity", &intensity,RI_NULL);
RiLightSource ("distantlight", "from", from, "to", to, RI_NULL);
RiSurface ("plastic", "Ka", &Ka, "Kd", &Kd, "Ks", &Ks, RI_NULL);
RiTranslate (1, 1, 1);
```

```
RiSphere (8, -8, 8, 360, RI_NULL);
RiWorldEnd ();
RiFrameEnd ();
RiEnd ();
}
```

```
Format 512 512 1
PixelSamples 2 2
FrameBegin 1
Display "t1.tif" "file" "rgb"
Projection "perspective" "fov" 45
Translate 0 -1.5 10
Rotate -90 1 0 0
Rotate -10 0 1 0
WorldBegin
LightSource "ambientlight" 1 "intensity" 0.5
LightSource "distantlight" 2 "from" [ 0 0 1 ] "to" [ 0 10 0 ]
Surface "plastic" "Ka" 0.5 "Kd" 0.8 "Ks" 0.2
Translate .5 .5 .8
Sphere 5 -5 5 360
WorldEnd
FrameEnd
FrameBegin 2
Display "t2.tif" "file" "rgb"
Projection "perspective" "fov" 45
Translate 0 -2 10
Rotate -90 1 0 0
Rotate -20 0 1 0
WorldBegin
LightSource "ambientlight" 1 "intensity" 0.5
LightSource "distantlight" 2 "from" [ 0 0 1 ] "to" [ 0 10 0 ]
Surface "plastic" "Ka" 0.5 "Kd" 0.8 "Ks" 0.2
Translate 1 1 1
Sphere 8 -8 8 360
WorldEnd
FrameEnd
```

While the RI Spec only discusses API calls for C/C++, such API 'bindings' have been created by RenderMan users for other popular languages such as Java, Perl, Python and Tcl. So you can write standalone programs in those languages that use the API calls and output RIB by running the programs.

Shaders

Shaders can be created in two ways. The first way is to type them in by hand, using a text editor or source editor such as emacs, vi, etc. This is like writing a program in any other language such as C++ or Java, where many programmers prefer hand-editing source files compared to using more structured development environments (IDEs). The second way to create shaders is to use a graphical interface that lets you "connect the blocks" by visually hooking up pieces of shader functionality to output RSL source. Popular shader creation environments include Slim (which comes with the Maya MTOR plugin) and ShaderMan, a free standalone shader generator.

Maps

A map is a pre-existing piece of data in the form of a file, which your shader can access for use in its calculations. The most common example is a texture map which is used for adding detail to surfaces (eg. a floral pattern on to a shirt). What are the sources of maps? Texture maps can be a hand-painted image, a scanned piece of artwork or even a digital

photo (image processed or used as is). Many other types of maps are generated by RenderMan itself, in a prior 'pass'. The idea is to render the scene once to create a map, and render it again to create the desired output, taking into account the map created in the previous step. For example shadows can be created this way, by first creating a shadow map with a first pass and then rendering again to use the shadow map to compute a shadow.

To use images as texture maps, they need to be pre-processed (usually this means converting them to an image pyramid or MIP map). This is done with a texture creation program that ships with the renderer. For example in PRMan, a simplified invocation of the texture generator is this: txmake flower.tiff flower.tex The input to the txmake program is a standard TIFF image, and the output is a new .tex texture file which can then be refered to in a shader, using a texture() RSL function call.

Here is a list of maps used in PRMan. Non-PRMan renderers can also use many of these - consult your documentation to see which ones are applicable in your case.

- texture maps - used to add surface detail, and to colorize surfaces, add pre-computed lighting, affect transparency, etc.
- environment maps - these are used to mimic very shiny materials that reflect their environment
- reflection maps - to fake flat reflections, it is customary to render the scene from behind the reflecting surface (eg. a mirror) and reuse the 'reflection map' by projecting it on to the mirror in screen space
- normal maps - these are used with low resolution polymeshes to alter vertex normals, thereby giving the appearance of a high resolution model
- shadow maps - these are used to create shadows
- deep shadow maps - these contain more data than standard shadow maps, enabling richer-looking shadows (eg. colored, partially transparent shadows that exhibit motion blur)
- photon maps - used to create caustic patterns
- irradiance caches - useful for creating subsurface scattering effects
- occlusion maps - useful for creating visually-rich ambient lighting
- brick maps - these are in a PRMan-specific format (a form of tiled 3D MIP map), useful for storing calculations related to radiosity

Ways to extend RenderMan

- shaders can be considered "appearance plugins" since they are external programs which the renderer invokes, to carry out shading calculations.
- the 'Procedural' RIB call (and the corresponding RiProcedural() API call) make it possible to write standalone programs and DSOs (dynamically linked plugins) which can output RIB fragments. This facility can be used to create custom geometry (eg. fire sprites, foliage, characters..) and to render specialized primitives which are not described in the Spec.
- 'display drivers' DSO plugin files can be used to output the rendered pixels using custom image formats.

RIB - syntax, semantics

In this section we will take a brief look at the syntax of RIB files and how scene data is organized in a typical RIB file.

A RIB file contains a sequence of requests to the renderer, organized in the form of a keyword, often followed by additional data. Some of the data are specified as attribute/value pairs. Here are some examples:

```
Projection "perspective" "fov" 25
```

```
Rotate 19 0 0 1
```

Lines beginning with a # denote a comment and are ignored by the renderer, while lines beginning with a ## are renderer 'hints'. Examples:

```
# Correct for monitor gamma of 2.2
Exposure 1.0 2.2
```

```
##Shaders PIXARmarble, PIXARwood, MyOwnShader
##CapabilitiesNeeded ShadingLanguage Displacements
```

RIB files do not contain loops, branches or function declarations. In other words, it is not a programming language. It is more like a declaration language for specifying scene elements.

RIB files typically contain attributes and transformations expressed hierarchically, in the form of nested blocks (using AttributeBegin/AttributeEnd pair of keywords or TransformBegin/TransformEnd). Examples:

```
TransformBegin
 Translate -1.3 1 0
 Scale .5 .5 .5
 TransformBegin
  Color [.22 .32 .58]
  ConcatTransform [1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
  PointsGeneralPolygons [1] [3] [2 1 0] "P" [0 0 0 1 0 0 1 0.7265 0]
 TransformEnd
TransformEnd
```

```
AttributeBegin
Surface "rmarble" "veining" 5 "Ka" 0.25 "roughness" 0
Color [.96 .96 .04] #[0.52 .52 .58]

TransformBegin
# Translate 0 0 0
Disk 0 0.5 360
TransformEnd

TransformBegin
Translate 1 0 0
Disk 0 0.5 360
TransformEnd
AttributeEnd
```

By nesting transforms and attributes this way, complex geometry and material setups can be specified. Underlying this hierarchical specification are the notions of a graphics 'state' and a 'current transformation matrix', and the attribute/ transform blocks manipulate these by pushing and popping data off stacks that maintain the current state.

Geometry, lights and materials are specified inside a WorldBegin/WorldEnd block, while camera and image parameters come before the WorldBegin:

```
Format 750 750 1
Display "SquareSquaresRel.tiff" "framebuffer" "rgb"
Imager "background" "color" [.57 .22 .13]
Projection "perspective" "fov" 90
```

```
Projection "orthographic"
ScreenWindow -20 20 -20 20
LightSource "distantlight" 1
Translate -16 -16 0
Translate 0 0 1
WorldBegin
AttributeBegin
# 0,0
Color .25 .25 .25
Scale 18 18 1
Polygon "P" [0 0 0 1 0 0 1 1 0 0 1 0]
AttributeEnd
AttributeBegin
# 18,0
Color .5 .5 .5
Translate 18 0 0
Scale 14 14 1
Polygon "P" [0 0 0 1 0 0 1 1 0 0 1 0]
AttributeEnd
AttributeBegin
# 18,14
Color .75 .75 .75
Translate 18 14 0
Scale 4 4 1
Polygon "P" [0 0 0 1 0 0 1 1 0 0 1 0]
AttributeEnd
WorldEnd
```

In the above example, the Display statement specifies a framebuffer "driver" (destination) for the output image. Image resolution is declared in the Format statement.

RIB files do not include shader files in them. Rather, they call out (or reference) shaders, which are separate from the RIB file and must be accessible to the renderer. Here are a couple of examples of shader specification:

```
# surface shader specification
Surface "wood" "Ka" .1 "Kd" 1. "grain" 12 "swirl" .75 "darkcolor" [.1 .05 .07]
# ...
# displacement shader call
Displacement "noisydispl" "ampl" .06 "freq" 5
# ...
```

RIB files can include other files (usually containing blocks of data in RIB syntax) using the ReadArchive statement. This is good for reusing assets (which can be kept in separate files ready for including in a master scene file), thereby keeping scene file sizes small.

```
WorldBegin
LightSource "distantlight" 1
LightSource "ambientlight" 2
Opacity [0.5 .5 .5]
Surface "showN"
# Costa_semi.dat contains polymesh data for a section of the Costa minimal surface
ReadArchive "Costa_semi.dat"
WorldEnd
```

Note that the RIB specification does not have a provision for specifying time-varying parameters, ie. animation curves. This means that for an animation sequence, each frame should be a self-contained block of RIB capable of producing an

output image. While the RIB spec. does allow for multiple frames of data to be stored in a single RIB file (using FrameBegin/FrameEnd blocks), it is more common for a RIB file to contain just one frame's worth of data. This allows RIB files for an image sequence to be distributed to multiple machines on a render farm, where each machine receives a single RIB file from the sequence and generates a single image from it. This also means that RenderMan renderers do not output animations in movie formats such as MPEG or Flash video. Animations are simply sequences of still images. The stills need to be post-processed using compositing and editing programs to create movie files for television, DVD production, etc. For movie production, post-processed stills (eg. composited with live action footage) are output to film using a film recorder.

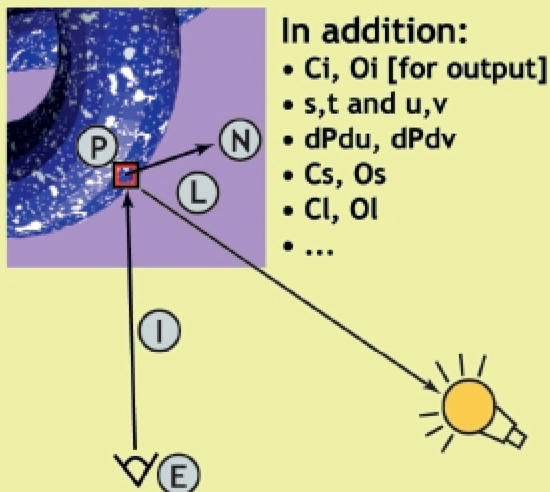
Shader writing

Here are some useful things to know about writing shaders. As mentioned before, RenderMan's powerful shading language (RSL) lets users write custom programs called shaders to completely define surfaces and their interaction with light sources. These shaders are referenced in RIB files where they are used to specify materials, light sources, etc. In that sense, shaders can be thought of as RIB plugins.

Highlights of RSL syntax:

- C-like language.
- types include float, string, color, point, vector, normal, matrix; these can be arrays too.
- usual set of operators, eg. +,-,*,/,%,== etc.
- control statements include if() and for().
- a rich collection of built-in functions that include mathematical, trigonometric, lighting, color transformation and map access calls. It is this powerful function library that makes RSL shader-writing a joy. Consult the Spec and your renderer's manual for a list of available functions.
- user-defined functions can be used to isolate and reuse blocks of code.
- computation-expensive calls can also packaged into "shadeops" which are compiled DSOs (eg. custom noise functions, Fourier transform code etc. can be coded up as shadeops). This is mostly done to gain execution speed.
- preprocessor directives are available, eg. #include, #define, etc.

You would write an RSL shader "blind", ie. without knowing exactly where on a surface it is going to invoked, how many times or in what order. Also, you do not have access to information about geometry that surrounds your shading neighborhood. What is available to you in the form of global variables is detailed information about the current point being shaded, for instance, its location, surface normal, texture coordinates, etc.



Here is a list of global variables pertaining to surface shaders. Consult the Spec for similar lists for other shader types.

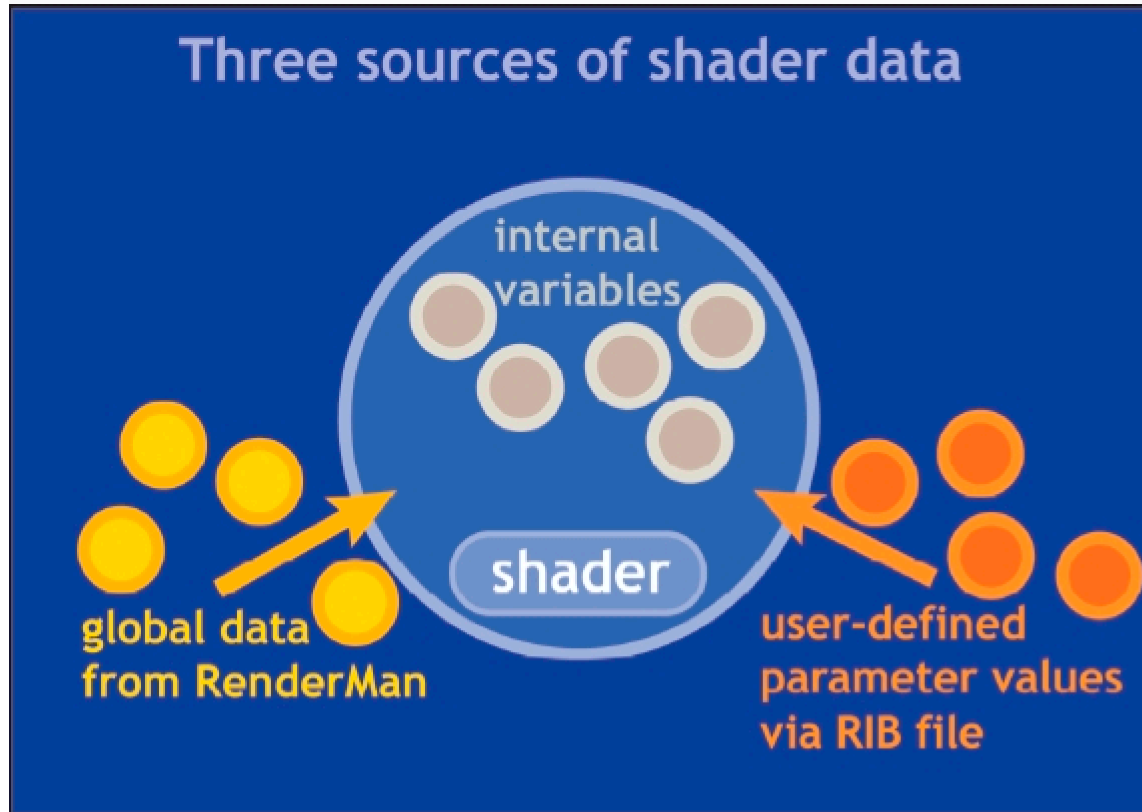
Information based on the RiSpec 3.2. Copyright Pixar Animation Studios

Surface Shader Variables

Variable Name	Type	Storage class	Description
Cs	color	varying	Surface Color described on the RIB file
Os	color	varying	Surface opacity described on the RIB file
P	point	varying	Position of shaded surface
dPdu	vector	varying	Derivative (tangent) of the surface position along u
dPdv	vector	varying	Derivative (tangent) of the surface position along v
N	normal	varying	Surface shading normal
Ng	normal	varying	Surface geometric normal
u,v	float	varying	Surface parameters
du,dv	float	varying	Change in surface parameters
s,t	float	varying	Surface texture coordinates
L	vector	varying	Incoming light ray direction*
Cl	color	varying	Incoming light ray color *
Ol	color	varying	Incoming light ray opacity*
E	point	uniform	Position of the eye or camera
I	vector	varying	Incident ray direction. Direction vector going from
			the camera to the current shading point
ncomps	float	uniform	Number of color components
time	float	uniform	Current shutter time
dtime	float	uniform	Amount of time covered by this shading sample
dPdtime	vector	varying	How the surface position P is changing per unit
			time, as described by motion blur in the scene.
Ci	color	varying	Shader output color
Oi	color	varying	Shader output opacity

* Available only in illuminance statements

In addition to the global variables, data can come into your shader via its argument list, where values for the parameters in the list are specified in the RIB call(s) for your shader. Also, data can be calculated, stored and used inside a shader via local variables.



As for shader creation tools, these have already been mentioned. With Maya, you can use Slim or MayaMan. Standalone shader generators include ShaderMan and Shrimp. The Air renderer comes with Vshade, a visual tool for creating shaders. But the most popular method of creating RSL shaders is to type them in source code form into a text or programmers' editor. The resulting source file needs to be compiled in order for the renderer to be able to load and invoke the shader. Eg. for PRMan, the compilation command is shader <myshader.sl> The result will be <myshader.slo>, which needs to accessible by PRMan when it renders a RIB file that makes a call to <myshader>.

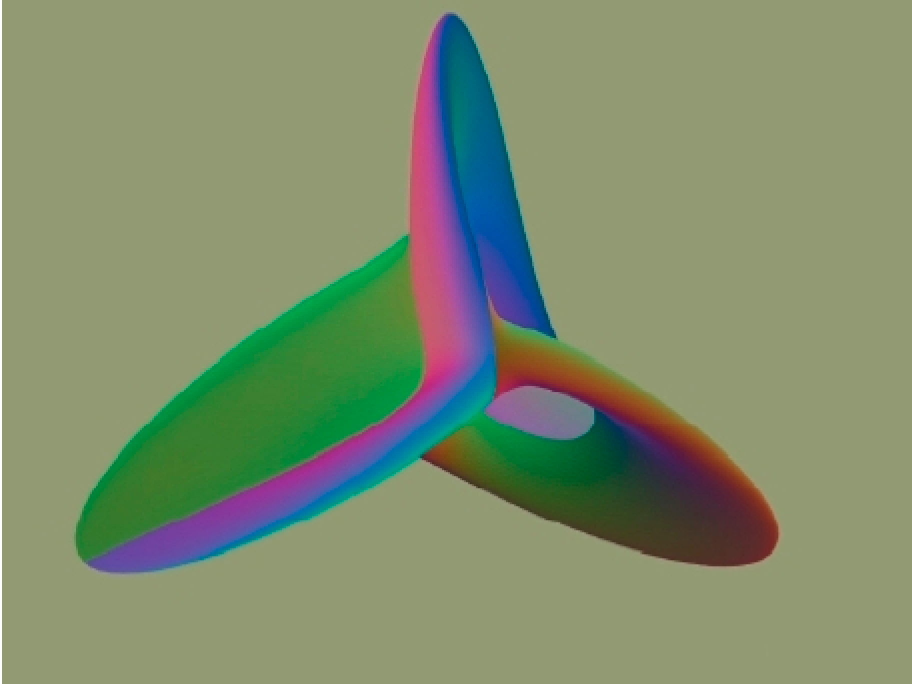
With RSL you can create five types of shaders:

- surface
- displacement
- light
- atmosphere
- imager

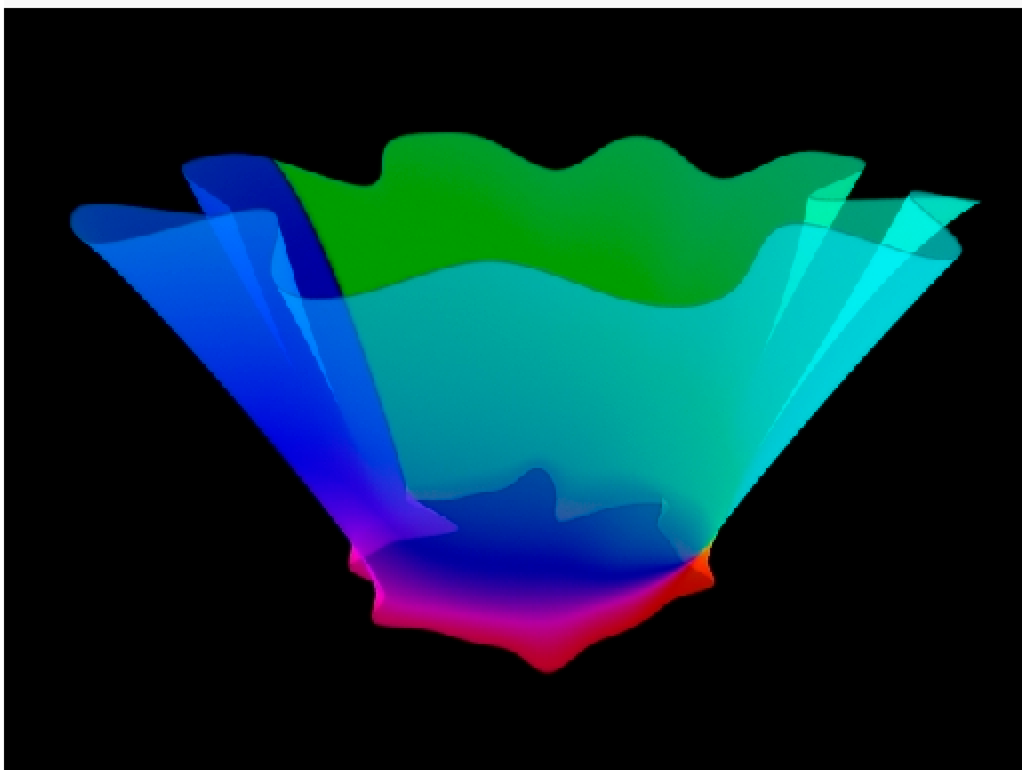
Here are six shaders (and sample results) to give you a taste for RSL.

```
surface showN()
{
  point PP;
  normal Nf;
  vector V;
```

```
  color Ct, Ot;


  normal NN = normalize(N);
  vector posNorm = 0.5*(vector(1,1,1)+NN);
  color posCol = color(comp(posNorm,0),comp(posNorm,1),comp(posNorm,2));
  color posGCol = 0.25*color(comp(posNorm,1),comp(posNorm,1),comp(posNorm,1));
  Oi = Os;
  Ci = posCol;
}
```



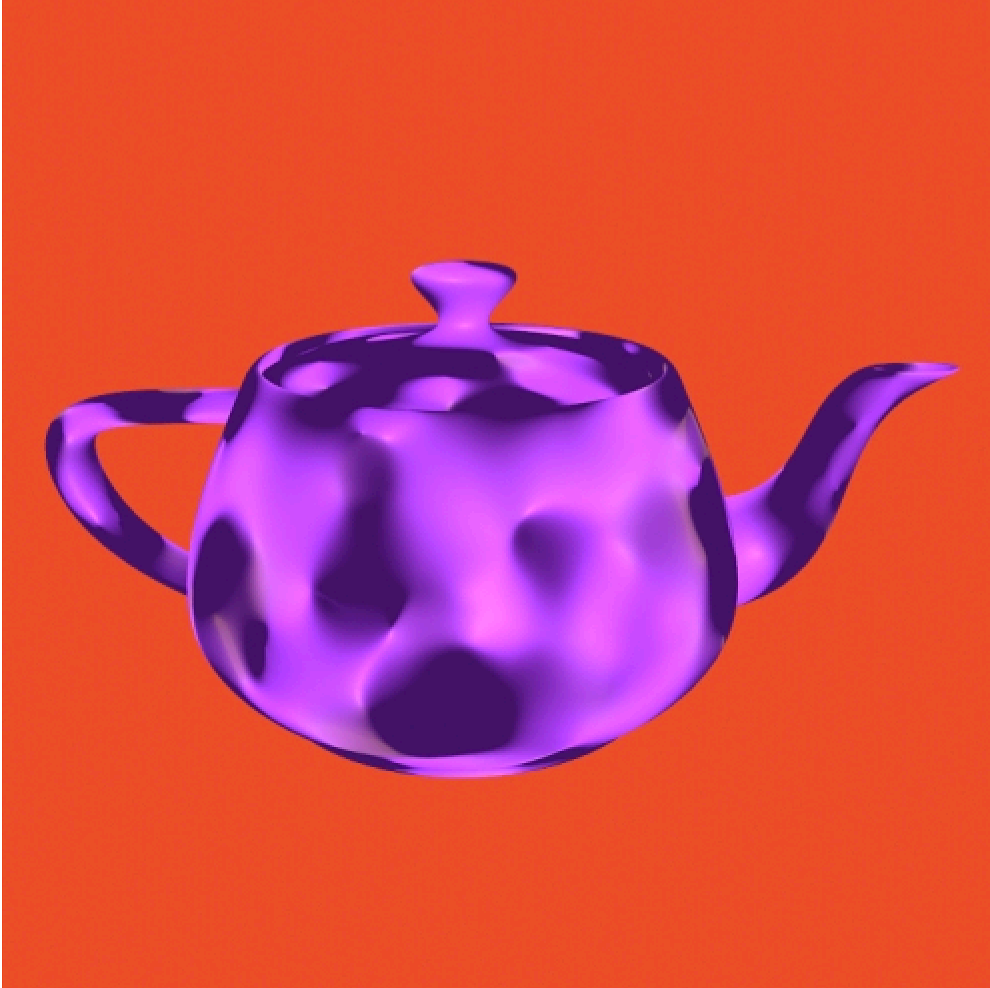
```
/* Straightforward texture lookup */
surface tex(string tmap="generic.tex";)
{
  float alpha;

  /* get base color from map */
  if(tmap!="")
  {
    color Ct = color texture(tmap,s,t);
    alpha = texture(tmap[3],s,t);
    Oi = alpha;
    Ci = Oi*Ct;

  }

}
```



```
displacement sinewaves(float freq=1.0, ampl=1.0, sphase=0, tphase=0, paramdir=0)
{
  // displace along normal, using sin(s) or sin(t) or both
  if(0==paramdir)
    {
      P += ampl*sin(sphase+s*freq*2*PI)*normalize(N);
    }
  else if (1==paramdir)
    {
      P += ampl*sin(tphase+t*freq*2*PI)*normalize(N);
    }
  else
  {
      P += ampl*sin(sphase+s*freq*2*PI)*sin(tphase+t*freq*2*PI)*normalize(N);
  }
  N = calculatenormal(P);
}// sinewaves
```

```
light
Kessonlt(
    float  intensity=1 ;
    color  lightcolor=1 ;
    float freq=1.0, coneangle=PI/2;
)
{
```

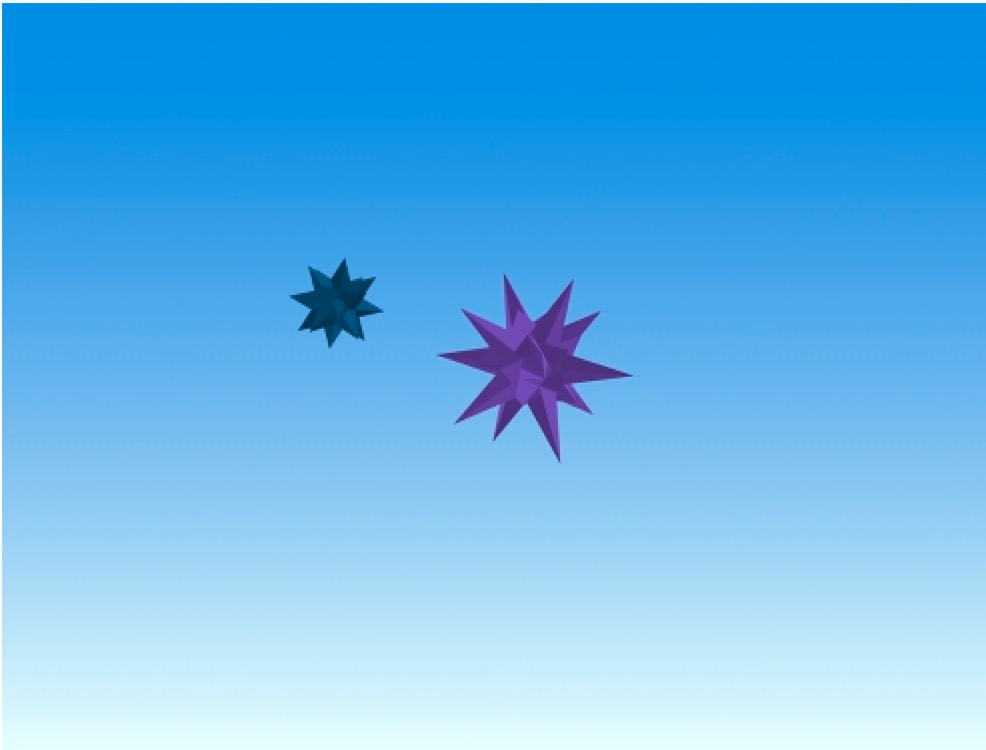
```
  point Pt = freq*transform("shader",Ps);
  vector ldir = 2*noise(freq*Pt) - 1;
  solar(ldir,coneangle)
    {
      Cl = intensity * lightcolor;
    }
}
```



```
volume underwater(
    float mindist=0, maxdist= 1;
    color fg=1, bg=1;
    float inten=1, gam=1, mixf=0.5;
    )
{
    color c;
    float d;

    d = length(I);
    if(d<=mindist)
      c = fg;
    else if(d>=maxdist)
      c = bg;
    else
      {
```

```
        d = (d-mindist)/(maxdist-mindist);
        d = pow(d,gam);
        c = mix(fg,bg,d);
      }

    Ci = inten*mix(Ci,c,mixf);
    Oi = mix( Oi, color (1,1,1), d );
}// underwater()
```

```
imager Imager_ramp
(
 color ctop = color(1,1,1);
 color cbot = color(0,0,0);
 float gam=1.0;
 )
{
 float curr_y;
 float rez[3];
 color rampcol;
 float mixf;

 option("Format",rez);
 curr_y = ycomp(P)/ rez[1]; // 0 to 1, top to bottom
 curr_y = pow(curr_y,gam);

 rampcol = mix(ctop,cbot,curr_y);
 Ci += (1-Oi)*rampcol;
 Oi = 1.0;

}
```



Resources

Since RenderMan has been around for a while, there are a lot of resources available to you for further exploration. Here are the main ones.

Books

- The RenderMan Companion - this was the first-ever RenderMan book. The first half deals with the C API for generating RIB, while the second half discusses shader-writing.
- Advanced RenderMan - lots of valuable info. on a variety of topics including lighting techniques and shader anti-aliasing.
- Essential RenderMan Fast - an introductory book dealing with the C language API, shading language and RIB.
- Rendering for Beginners - another introductory book that documents RIB calls and has sections on cameras, image quality and shader-writing.
- Production Rendering - written for software developers, this book is a collection of good ideas for implementing a modern film quality renderer.
- Texturing and Modeling - this book is about RenderMan per se but does use the RSL too illustrate many examples.

Notes

- SIGGRAPH course notes - ever since RenderMan's inception there have been courses held at SIGGRAPH on various aspects of RenderMan, including shader-writing, global illumination, use in production, etc.
- Steve May's RManNotes - a nice collection of lessons on shader-writing, using a layering approach. RManNotes is at http://accad.osu.edu/~smay/RManNotes/rmannotes.html
- Larry Gritz's 'RenderMan for Poets' - a small document (19 pages) which lists and explains the basic procedural API and corresponding RIB calls. You can find it at http://www.siggraph.org/education/materials/renderman/pdf_tutorial/poets.pdf.

- Malcolm Kesson's detailed notes on RIB and RSL, at http://www.fundza.com.

Forums, portals

- http://www.renderman.org - this is the RenderMan Repository ("RMR"), containing a ton of information and links. Here you can find a lot of shader source code and also links to PDFs of past SIGGRAPH course notes.
- http://www.rendermania.com - News and info. about RenderMan, and a rich collection of links to other people's pages and sites.
- http://www.rendermanacademy.com - excellent PDF tutorials focusing on RIB and shader writing.
- Google Newsgroup - comp.graphics.rendering.renderman - a place to post questions and to read the c.g.r.r FAQ maintained by Larry Gritz.
- http://www.highend3d.com/renderman - another place for downloading shaders and posting questions.
- http://www.deathfall.com - a broad CG 'portal' with an active RenderMan section

In addition to the above, you can find more information/knowledge at these sites for specific RenderMan implementations:

- PRMan: http://www.pixar.com
- 3Delight: http://www.3delight.com/
- Air: http://www.sitexgraphics.com/
- Angel: http://www.dctsystems.co.uk/RenderMan/angel.html
- Aqsis: http://www.aqsis.com
- Pixie: http://www.cs.utexas.edu/~okan/Pixie/pixie.htm
- RenderDotC: http://www.dotcsw.com/

Previous - Welcome and Introduction - Next What The RiSpec Never Told You

Retrieved from "http://rendermanacademy.com/sig2006/index.php/A_Brief_Introduction_to_RenderMan"

- This page was last modified 17:25, 9 Apr 2006.
- Content is available under GNU Free Documentation License 1.2.